The SOA Magazine

Feature Article

Message Auditing in Service-Oriented Architecture

by Kanu Tripathi

Published: December 17, 2008 (SOA Magazine Issue XXIV: December 2008)

Download this article as a PDF document.

Abstract: This article explores the subject of message auditing in a typical service-oriented system implementation. A case is built for having a system for auditing messages, followed by an exploration of the requirements this type of system can address. A candidate architecture and design of the system is the provided, concluding how the original goals can be attained. Finally some examples are given and some caveats are highlighted.

Introduction

Any serious service-oriented system implementation can be comprised of a number of services assembled into compositions. When you have a maze of services being called from many different clients, it can easily result in a formidable amount of information being passed around in form of messages. If principles of service-oriented design are followed, these messages would be in a standardized format. leading to a variety of possibilities as to how mechanisms can be positioned to filter and persist the messages and to also extract business intelligence from them in an arbitrary manner. The impact of standards-based messaging on auditing is as deep and as sweeping as its impact on the field of integration in general.

A Case for Auditing Messages in Service-Oriented Systems

Not much attention has been given to auditing messages in service-oriented eco-systems. Most modern infrastructure and service bus platforms provide simple message auditing mechanisms, for example, by writing messages to a log file or in a proprietary database. Typically, these mechanisms are not optimized for performance and are recommended for diagnostic purposes only.

This traditional ignorance towards auditing is most likely due to the fact that auditing is usually not seen as primary business need. Also, the requirements for auditing can be so daunting and unpredictable that it can be very difficult to comprehend and achieve in a consistent manner, especially across heterogeneous environments.

With the advent of service-orientation, service design principles, standardized message structures and commercial infrastructures that directly enable service-oriented computing, it is becoming significantly easier to build a system that is generic enough to plug into a variety of platforms and also specific enough to capture only certain messages on an "as need" basis. As a result, we can now build message auditing systems without having to creep inside of and customize the service logic.

This effectively enables us to:

- respond to arbitrary queries pertaining to compliance requirements
- gather business intelligence from arbitrary perspectives by running queries against persisted messages
- set up observation systems to raise alarms when certain events happen

- extract diagnostic information about systems to better optimize their resources
- create a dashboard to observe the overall state of systems in real-time

Let's now explore these requirements in more detail and then establish a high-level design and implementation.

Typical Requirements

Any reasonable service-oriented implementation can have multiple services (and even multiple versions of services) along with multiple XML schemas. The services may be implemented on more than one platform and may be called by diverse clients. The volume and size of messages are not predictable – because well-designed services are typically interoperable, reusable and composable, we cannot predict how and when new usages of existing services will emerge.

This leads to the following requirements that a message auditing system may need to address:

Flexibility to Handle Messages that may have Different Structures

The auditing system cannot assume any specific structure or schema for the messages. A request or response message at one service may look very different from the request or response at another service. Even different versions of same service may have different message structures.

Ability to Specify Criteria to Filter Messages so that only Specific Messages are Audited

The message auditing system must support message filtering to specify which messages really need to be audited. It is not incomprehensible to visualize that not all messages need to be audited. An ability to filter the messages reduces the load on the auditing system and creates a meaningful message warehouse, which can be easily managed and processed to gather on-going business intelligence. The challenge in this requirement is that since the messages can be of any structure, the criteria for filtering can also vary.

Ability to Map messages to the Service Instances from which they Emanated

The message auditing must be able to support the mapping of an audited message to the instance of the service from which it originated, so that, in case more details have to be found, the log files on the server may be referenced. In the absence of this type of mapping logic, it may be difficult to gain the needed amount of insight into how message were previously processed.

Ability to Support Multiple Destinations where Message need to be Stored

The message auditing system must be able to support multiple destinations for the audited messages. Typical examples of where the message may be sent are: the database, e-mail, files and folders, queues and other services. The database is commonly considered the single most important destination, since it is permanent and can be used to generate business and reporting intelligence. However, it is important to support other destinations for real-time reporting, events, and for help with diagnosing problems.

Ability to Scale to Accommodate Increasing Load

As mentioned previously, because it is impossible to predict how a service eco-system will grow and evolve, it is also not possible to accurately estimate future loads placed upon the message auditing system. Therefore, the system needs to be inherently and seamlessly scalable so that any required infrastructure upgrades can be added without affecting existing services.

Must be Not be Intrusive to Service Logic

An auditing system should not impose significant change upon existing services. Services must be designed so that their autonomy is maximized while their coupling to the infrastructure is minimized. When incorporating a an auditing system (be it accessed directly by service logic or made available via separate utility services), care needs to be taken so that it should not introduce negative service coupling requirements.

Must be Autonomous

Message auditing systems should not be dependent on services. They may depend on the standardized schemas being used to define service contract types (following the service design principle of Standardized Service Contract), but they should not have any direct dependencies on business services, their message structures, or service logic. This enables auditing services (or systems) to exist as independent parts of the enterprise with their own life cycles.

Must Act as an Optional Feature

Not all services need auditing and those that do may not need auditing all the time. Message auditing logic must therefore be designed to be as "pluggable" as possible, so that it can be easily enabled and disabled without affecting services and their supporting infrastructure.

Must be Flexible and Extensible

The auditing requirements can be so different from one organization to another or even from one service inventory to another in the same company, that the auditing system should not be built to address a predefined set of auditing requirements. Instead, the system must act as a platform which can be customized or extended to accommodate evolving requirements.

Ability to Define Arbitrary Queries for Reporting or Event Generation Purposes

Ad-hoc reporting is often ignored in traditional auditing systems. In order to support this requirement, messages must be stored in a manner that makes it is possible to query them to generate both reports and events. Keeping in mind that each service operation has its own message structure, it must be possible to specify the query in an arbitrary manner. This is also due to the fact that the requirements for reports and events cannot be predicted since the auditing on a system may lead to the need for ad-hoc queries.

Ability to Generate Automated Reports

It has become increasingly common for systems to comply to regulation-based auditing requirements. With the advent of Sarbanes-Oxley Act in particular, companies are required to provide high-quality reporting capabilities in order to respond to auditing-based compliance queries.

Solution

There are several SOA infrastructure platforms that simply have insufficient auditing capabilities. Building your solutions on such a foundation may force you to later adopt a third-party auditing solution. However, auditing requirements can become so varied that it simply may not be feasible to build or use one system that can address them. Therefore, you need to place reasonable assumptions and constraints on what a given auditing system can and cannot do.

For example:

- Service instances must be identifiable by some unique ID that can be stored as metadata along with the messages.
- The auditing system should not be responsible for decrypting messages.
- The auditing system may be allowed to add its own information as headers to the messages.
- It must be possible to identify the version of service, operation and schemas being used from the message.

To limit the scope of a message auditing system, the following points must also be considered:

- Auditing logic should not be used for routing purposes. The message processing chain should be part of service-logic.
- Sometimes, the word "auditing" is used to describe a business requirement for applications. For example, in a customer management application, the administrator must be able to see the history of

how customer information was modified and by whom. This should not be confused with "message auditing", which is logic that is considered to be part of an agnostic system (a system that is not bound to a specific business context).

- Auditing systems may alter messages by adding their own header information. This should not be considered as a "breakage" in the message integrity because these headers are generally used for auditing purposes only.
- The message store should not be used for non-repudiation. This is again is due to the fact that headers may be added to the message for auditing.

Architecture and Design

There can be many different ways to build an auditing system. Here, we will explore one possible approach based on the context and requirements discussed so far.

Overall the solution can be broken down into the following parts:

1. Service Instance Manager
2. Message Interceptor
3. Filter Manager
4. Filters-Service Instance Mapper
5. Destination Manager
6. Filter-Destination Manager
7. Report Manager
8. Report Scheduler

1. *Service Instance Manager*

   The message auditing must be done in a way that it should be possible to track from which instance of what service the message was captured. So, it is important to have a mechanism in place that consistently and uniquely defines each service instance. It may also be a good idea to implement a self-registration process so that when an instance of a service is started, it can register itself in a database.

2. *Message Interceptor*

   The capturing mechanism must be pluggable so that it is easily enabled or disabled. It has to be non-intrusive to service logic and to enable high levels of scalability, it should be designed to simply capture and send messages to a queue.

   This module is also responsible for gathering metadata about the message (e.g. the service instance identifier to map the messages to the service log files). All of this metadata must be either added to the messages via headers or it may be captured as separate fields.

   The module must be able to capture request and response messages together and send them as pairs for auditing. This ensures that request and response message information will be bundled for future processing and reporting purposes.

   This module should not apply filters to the messages since that may introduce time-consuming logic that could adversely affect the runtime service performance. Typically, a message capturing mechanism can be developed by implementing message handlers or interceptors.

3. *Filter Manager*

   A filter is a simple function that would take request and response messages and associated metadata and determine whether a given set of messages needs to be audited or not. Typically a filter would look into the message content and apply a rule to find out if certain conditions resolve to true or not.

   Some common types of filters that can be implemented and readily used are

   - look for a certain string in the message to be present or absent

- apply a given XPath query to the message that results in a Boolean response
- apply a given XQuery query to the message that results in a Boolean response
- apply interfaces to define arbitrary criteria based on the message content

While defining the filter, it must be indicated whether to apply it on the request or the response. The definitions of these filters may be created and stored in a database or configuration file.

4. *Filter-Service Instance Mapper*

   A mechanism to map the filters with service instances is needed to determine which filters to apply for a given set of messages and their metadata. This can also be done in a database or a configuration file. Note, however, that a database may be more suitable do to the need to sometimes establish many-to-many relationships.

5. *Destination Manager*

   The destination for filtered messages represents the location they need to be sent to for storage. Some possible destinations can be the database, a queue, e-mail, files and folders, URLs, other services, or any implementation of a simple interface that can accept the request, response, message metadata and filter and is able to send them to whatever destination they need to go. All destinations may be stored in configuration file or a database.

6. *Filter-Destinations Mapper*

   It needs to be specified for each filter where the messages must be sent. This can be achieved by specifying a list of destinations for each filter register.

7. *Report Manager*

   The database is a mandatory destination if it is important to run reports on audited messages. Defining reports at a basic level would require setting up query that needs to be executed to select only relevant messages and further provide some way to render these messages. A simple yet powerful solution for this is storing the messages in XML type columns and specifying queries in terms of XPath or XQuery queries. As mentioned previously, the messages may be based on an arbitrary structure and it may therefore be easy to specify rendering formats with technologies like XSLT.

8. *Report Scheduler*

   This is a regular scheduling mechanism that can run reports based on the provided schedule and report definition.

Putting It All Together

Figure 1 identifies some of the interfaces that will enable the creation of the previously listed solution parts. For the sake of brevity, only important interfaces are listed.

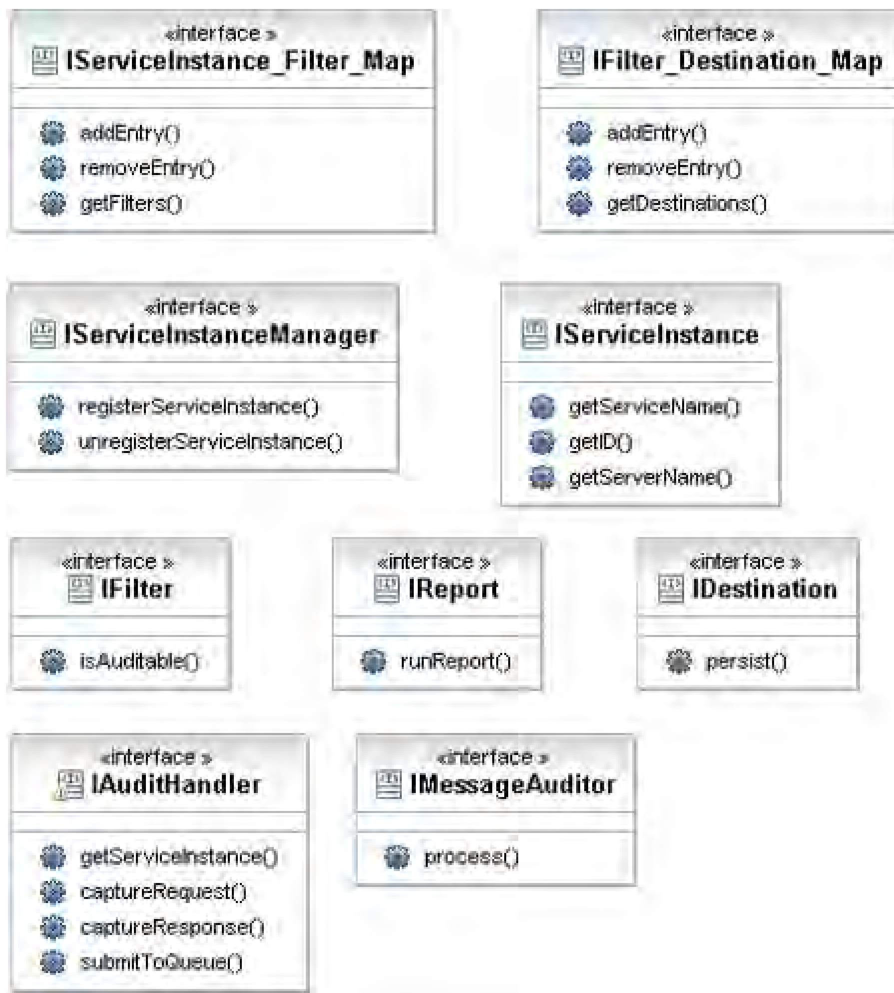

Figure 1

Once all these components are in place, we can study how the system will work at runtime and how the original requirements can be met. At service start-up, the service instance must register itself with a central registry as displayed in Figure 2.

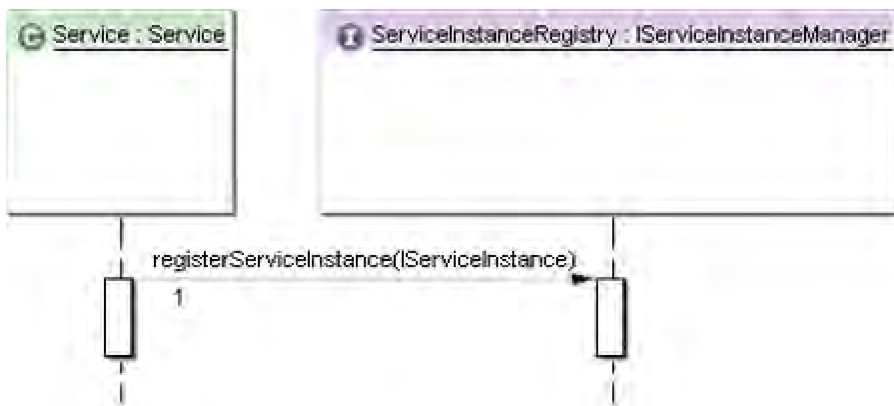

Figure 2: At service startup, the service instance object is created and registered

The following steps demonstrate how the auditing system would work at runtime:

1. A service client sends a message to a service instance.
2. The auditing interceptor captures the service instance identity, request, and its metadata, and stores this information locally.
3. The service logic processes the request and constructs the response.
4. The auditing interceptor captures the response.
5. The auditing interceptor combines the request, response, metadata, and service instance identity, and

sends this combined packet to a queue.

6. The response is returned to the client.

These steps are illustrated in Figure 3.

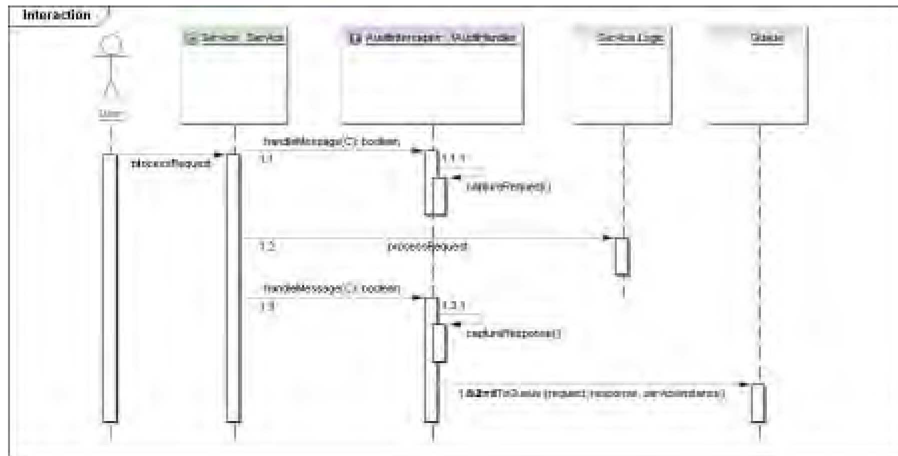

Figure 3: This figure shows how audit interceptor captures the request and response and simply submits to a queue. This makes the auditing system less intrusive and more scalable. Note that filters are not applied at this point to reduce the additional processing.

[view larger image]

It evident that the auditing interceptor acts in a non-intrusive and scalable manner. It does not depend on message structures and individual service instances.

Let's now see how the messages can be processed by the queue listeners:

1. The queue listener picks the message packet that has request, response, service instance identifier, and other message metadata.
2. The listener finds the mapped filters for given service instance.
3. The filters are applied to the messages and, as a result, each filter indicates whether the message was a match or not. It is possible that many filters are provided for a given instance of the service and more than one may indicate a match.
4. If no filter is specified or no filter indicates that the message must be audited, the message is simply discarded.
5. For every filter that indicates that the given set of messages must be audited, it is determined what the destinations for that filter are.
6. The message packet is then sent to every destination mapped for the given filter.

These steps are further illustrated in Figure 4.

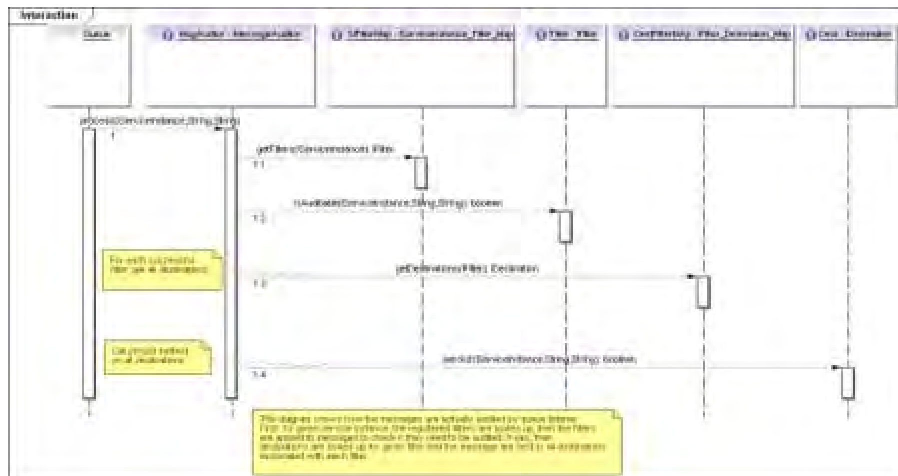

Figure 4: This diagram shows how the messages are actually audited by queue listener. First, for given service instance, the registered filters are looked up, then the filters are applied to messages to check if they need to be audited. If yes, then destinations are looked up for given filter and the message are sent to all destinations associated with each filter.

view larger image

The sequence of processing steps shown in Figure 4 represents a flexible processing model that allows for customizable behavior for each service instance and filter. It is extensible since any new filter may be added to the service instance and any new destination may be provided for each filter.
Once the reports have been defined using XPath or XQuery, the queries can be run against the messages. Since the message structure can be arbitrary, it is important to save them in XML type columns.

Let's now consider some sample cases and see how the auditing system can support these requirements:

1. You want to track all activities of a user

It is a common situation where you may want to track a user's activity (e.g. for the purpose of diagnosing a problem). Assuming that each message would have a user ID embedded in it, you can create an XPath filter and add that to all service instances. If you want to quickly diagnose the problem, you could specify your e-mail address as the destination. Once this filter is enabled, the messages will start coming to your inbox. After you have collected relevant messages, you can disable the filter.

2. You want to find how a lookup service is being used.

You might have a service that lets users lookup some service providers in a given area. You may want to find out how this is being used to improve user interface. You can capture all messages for this service and query using XQuery or XPath to count how many times zip, state or city were provided, what mile radius is usually selected by users, and so on. Based on this, the user-interface may pre-select some values.

Care must be taken to ensure that interaction with the queue is efficient (for example, it may be executed in a separate thread). If the messages hold sensitive data, the message database must be designed and managed so that the data in not compromised. Since the message auditing will simply store the messages without association with entitlements, it may not be possible to create reports based on some access control. Messages with attachments pose another challenge since the attachments themselves may also need to be stored and queried.

Conclusion

With the advent of XML-based technologies, the evolution of service design principles, service design patterns, and industry standards, message auditing has become viable commodity within the modern SOA eco-system. Adding auditing logic to your systems opens up a flood gate of opportunities for extracting business intelligence from messages.

The beauty of message auditing with XML is that it does not bind you to a specific data structure and therefore allows for the extraction of information in an arbitrary manner. This can be the best defense against the random nature of service-oriented system auditing, which can span from performance data to compliance requirements, from user behavior to drawing business intelligence, and many more usages.

References:

[REF-1] "SOA Principles of Service Design", by *Thomas Erl (Prentice Hall)*.
[REF-2] "Web Service Contract Design and versioning", by *Erl et al (Prentice Hall)*.
[REF-3] Handler API on JEE 1.5 platform,
https://java.sun.com/javaee/5/docs/api/javax/xml/ws/handler/Handler.html
[REF-4] http://en.wikipedia.org/wiki/XPath
[REF-5] http://en.wikipedia.org/wiki/XQuery
[REF-6] "Design Patterns: Elements of Reusable Object-Oriented Software" by *Gamma, Helm, Johnson, Vlissides (Addison-Wesley)*.

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL

 Copyright © 2006-2009
SOA Systems Inc.